Insight Knowledge in Search Based Software Testing

Andrea Arcuri
The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.
a.arcuri@cs.bham.ac.uk

ABSTRACT

Software testing can be re-formulated as a search problem, hence search algorithms (e.g., Genetic Algorithms) can be used to tackle it. Most of the research so far has been of empirical nature, in which novel proposed techniques have been validated on software testing benchmarks. However, only little attention has been spent to understand *why* meta-heuristics can be effective in software testing. This insight knowledge could be used to design novel more successful techniques. Recent theoretical work has tried to fill this gap, but it is very complex to carry out. This has limited its scope so far to only small problems. In this paper, we want to get insight knowledge on a difficult software testing problem. We combine together an empirical and theoretical analysis, and we exploit the benefits of both.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Theory, Reliability

Keywords

Evolutionary Testing, Theory, Object-Oriented Software, Search Landscape

1. INTRODUCTION

In software engineering there are many tasks that are extremely expensive. For example, it is very common that software testing can take up to half the resources of the development of a new software [4]. This is the reason why in literature there has been a lot of effort to automate as many of these tasks as possible.

In software testing several different techniques have been developed. Among them, the use of *search algorithms* like for example Genetic Algorithms [8] has been quite successful [13]. However, most of the research so far has been of empirical nature. This has validated the use of search algorithms on many benchmark problems in software testing. And it made it possible to test the effectiveness of new techniques aimed at improving the performance of search algorithms (e.g., *Testability Transformations* [6]). Unfortunately, in general these types of analysis have difficulties in explaining *why* a search algorithm is effective on a software testing problem and which are its potential and limitations.

Theoretical analyses have been recently started to be applied to search based software testing to try to get insight knowledge on how search algorithms work. We are aware of only few results, that are on computing unique input/output sequences for finite state machines [12, 11], the application of the Royal Road theory to evolutionary testing [7], and testing the triangle classification problem [2, 1]. Unfortunately, theoretical analyses are difficult to carry out, and that has limited their scope so far to only small problems.

In this paper, we want to get insight knowledge in a challenging testing problem. Because a full theoretical analysis would not be feasible, we integrate it with an empirical analysis to mitigate its limitations.

We choose to analyse the testing of *Red-Black Trees* [5]. In particular, we consider the white-box scenario in which the full coverage of all the branches of the software is sought. *Containers* are a typical benchmark in software testing of object-oriented programs. For example, they have been used to validate search algorithms [3], more traditional techniques [17] and hybrids of these two [9]. Empirically, it has been shown that red-black trees are the most difficult to test among the considered containers [17, 3, 9].

We analysed four different search algorithms: Random Search (RS), Hill Climbing (HC), (1+1) Evolutionary Algorithm (EA), and a Genetic Algorithm (GA). They are typical search algorithms used in the software testing literature.

The main contributions of this paper are:

- We deeply analyse a difficult search based software testing problem.
- We show how empirical and theoretical analyses can be combined together to obtain more insight knowledge.

The paper is organised as follows. Section 2 describes the search problem we want to solve. A description of the search algorithms that we use follows in Section 3. On one hand, in Section 4 we carry out a type of empirical analysis that is common in literature. On the other hand, in Section 5 we

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

do more detailed analyses. Finally, Section 6 concludes the paper.

2. THE SEARCH PROBLEM

We took an implementation of red-black trees from the class TreeMap in the the Java API edition 6. That class has several methods, and we concentrated only on two of them: `put(key,value)` and `remove(key)`. A test case is hence a sequence of function calls of these two methods. Let l be the length of this sequence. Each sequence is called on a new tree object initialised with its default constructor.

According to our previous work [3], for branch coverage it is enough to use integer objects for the inputs of these methods, as long as all the possible permutations of their relative order are permitted. To do that, we can choose input integers i in the range $0 \leq i < l$. Note that constraints on the input variables are commonly used to reduce search efforts, but in general it cannot be guaranteed that any global optimum would lie within those limits. Furthermore, we use the same objects for `keys` and `values` (so we can consider `put` like having only a single input), and for simplicity of the analyses we do not consider the insertion/removal of `null` objects.

Given length l, the search space S is composed of $|S| = 2^l l^l$ elements (there are 2^l possible sequences of the two different methods, and each method can have l different values for input). We chose to consider a fixed length $l = 7$, which results in a search space composed of 105,413,504 elements. This number is sufficiently high to make sensible the study of search algorithms on this problem, but not high enough to preclude some types of exhaustive analyses of the full search space. Each solution is composed of 14 variables, 7 for the methods and the other 7 for their inputs.

On one hand, fixing the length of the test sequence could preclude the execution of some of the code branches (i.e., the test sequence is too short). On the other hand, too long sequences would make the computational cost of the fitness function too expensive. Including the length of the sequence in the search problem would help to mitigate this issue, but it would make the analyses more difficult. We hence leave it for future work.

We exhaustively evaluated all the test sequences in S, and for each branch we kept track of how many test sequences execute them. The branch that has been executed fewest times, but at least once, belongs to the private method `fixAfterDeletion`. We call B this branch, and we chose it for our analyses because it is likely the most difficult to cover with conventional search algorithms (note that this is not necessarily true, because it depends on the search landscape and on the employed search algorithm). Figure 1 shows the part of code in which the branch B appears.

To use search algorithms to find a test sequence that executes B, we use a fitness function that is based on the commonly used *branch distance* and *approximation level* [13]. Note that `fixAfterDeletion` is a private method that cannot be called directly. If that method is not executed at least once, we give the worst possible fitness value. Otherwise, we calculate the branch distance and approximation level in the usual way.

The predicates on which the branch distance is calculated from are all booleans. This is a problem (known as the *flag problem*), because the search space will have large plateaus. The possible different fitness values are only 6. In the rest

```
private void fixAfterDeletion(Entry<K,V> x) {
  while (x != root && colorOf(x) == BLACK) {
    if (x == leftOf(parentOf(x))) {
      Entry<K,V> sib = rightOf(parentOf(x));
      if (colorOf(sib) == RED) {
        //TARGET, branch B
```

Figure 1: Part of the TreeMap code in which there is the target branch B.

of the paper, for simplicity we map these values to integers from 0 to 5 without changing their relative order. It is a minimisation problem of fitness function f, in which the fitness value 0 represents the fact that the branch B has been covered.

3. THE SEARCH ALGORITHMS

There exist several search algorithms with different names. In general, a name does not represent a particular algorithm. It rather represents a family of algorithms that share similar properties. In this paper, we analyse four different implementations of four different families of search algorithms. Obviously, the results on a particular implementation cannot be directly extended to its family.

A search algorithm might not find a global optimum in a reasonable amount of time. Therefore, it is common to put premature stopping criteria that are based on the available computational resources. A typical simple example is to put an upper bound to the maximum number of fitness evaluations that are allowed. In this paper, because in the empirical experiments all the search algorithms find global optima in reasonable time, for simplicity we do not put any premature stopping criterion, unless otherwise stated.

RS is the simplest search algorithm. Random solutions are sampled until a global optimum is found. The information given by the fitness function is only used to check whether a global optimum has been sampled. RS is commonly used in literature as a base-line for comparing other search algorithms. What distinguishes among RS algorithms is the probability distribution used for sampling the new solutions. We employ a uniform distribution. Its pseudo-code is shown in Figure 2.

HC starts from a search point, and then it looks at *neighbour* solutions. A neighbour solution is structurally *close*, but the notion of distance among solutions is problem dependent. If at least one neighbour solution has better fitness value, HC "moves" to it and it recursively looks at the new neighbourhood. If no better neighbour is found, HC re-starts from a new solution. HC algorithms differ on how the starting points are chosen, on how the neighbourhood is defined and on how the next solution is chosen among better ones in the neighbourhood. We choose the starting points at random. The neighbourhood N is defined by swapping one single method at one time (i.e., a `put` is replaced by a `remove`, and vice-versa) without altering the inputs, and by modifying each input by $\pm 1\% l$. Hence the neighbourhood has size $l + 2l$. The strategy δ to visit the neighbourhood starts from left to right in the representation (i.e., the method in the first function call is modified, then the second visited neighbour consists of adding 1 to the input of the first method, etc.). As soon as a better solution is found, HC moves to it. In visiting the new neighbourhoods, instead of starting

```
while global optimum not found
    Choose I uniformly at random from S.
```

Figure 2: Pseudo-code of RS

```
while global optimum not found
    Choose I uniformly at random from S.
    while I not a local optimum in N(I)
        Choose I' from N(I) according to strategy δ.
        if f(I') < f(I), then
            I := I'.
```

Figure 3: Pseudo-code of HC

from the leftmost variable, HC starts from the successive of the last considered (whose modification has led to a new better solution), and it continues toward the rightmost, for then coming back as in a ring to the leftmost and finishing moving right until all neighbours have been visited. The pseudo-code of HC is shown in Figure 3.

(1+1)EA is a single individual evolutionary algorithm. A single offspring is generated at each generation by *mutating* the parent. The offspring never replace their parents if they have worse fitness value. In a binary representation, a mutation consists of flipping bits with a particular probability. Typically, each bit is considered for mutation with probability $1/k$, with k the length of the bit-string. In our case, we have that a test sequence is composed of l function calls. We consider the methods in the test sequences as bits (e.g., 0 to represent `put` and 1 for `remove`), and the method inputs as groups of bits that are mutated together with a special operator. Therefore, in our case we have $k = 2l$. Each method type and input is mutated with probability $1/(2l)$. In case of a mutated input, a new different valid value is chosen with uniform probability. The pseudo-code is shown in Figure 4.

GAs are the most famous meta-heuristic used in the literature of search based software testing, and they are inspired by the Darwinian Evolution theory. They rely on four basic features: *population*, *selection*, *crossover* and *mutation*. More than one solution is considered at the same time (*population*). At each generation (i.e., at each step of the algorithm), some good solutions in the current population chosen by the selection mechanism generate offspring using the crossover operator. This operator combines parts of the chromosomes (i.e., the solution representation) of the offspring with probability P_{xover}, otherwise it just produces copies of the parents. These new offspring solutions will fill the population of the next generation. The mutation operator is applied to make small changes in the chromosomes of the offspring. To avoid the possible loss of good solutions, a number of best solutions can be copied directly to the new generation (*elitism*) without any modification. We used a rank-based selection [18] with bias 1.5. The employed crossover operator is a single point crossover. The mutation is done in the same way as for (1+1)EA. Figure 5 shows the pseudo-code of the employed GA.

```
Choose I uniformly at random from S
while global optimum not found
    I' := I.
    Mutate I'.
    If f(I') ≤ f(I)
        then I := I'.
```

Figure 4: Pseudo-code of (1+1)EA

Table 1: Results of empirical experiments. Each search algorithm has been run 100 times.

Algorithm	Min	Median	Mean	Max	Variance
RS	736	66170	82570	272700	$4.72 \cdot 10^9$
HC	102	30590	46770	227400	$2.15 \cdot 10^9$
(1+1)EA	141	2506	3927	15270	$1.21 \cdot 10^7$
GA	809	23030	43730	264900	$2.59 \cdot 10^9$

4. FIRST EMPIRICAL EXPERIMENTS

A common practice in literature is to run a randomised search algorithm on a testing problem several times (e.g., 100 times) and then collecting statistics from those runs. It is technically unsound to run a randomised search algorithm only once, because the result would be too noisy. All the search algorithms in this paper are randomised.

We ran this type of experiments on all the four considered search algorithms. In particular, GA has a population of 100 individuals, $P_{xover} = 0.75$ and elitism rate of 1 individual per generation. We have not tuned these three values, and they represent reasonable choices that are common in literature.

Table 1 shows the results of each search algorithm run 100 times with different random seeds. The performance of a search algorithm is based on how many fitness evaluations it requires before reaching a global optimum.

Mann-Whitney U tests with significance 0.05 have been carried out to see if there is any statistical difference among the median values of these search algorithms. Resulting p-values are shown in Table 2. It should be noted that we had to use a non-parametric statistical test because we do not have sufficient evidence to suppose any particular distribution of the data.

The experiments show that HC and GA are roughly twice as fast as RS. There is no statistically significant difference between HC and GA. The fastest search algorithm is (1+1)EA, and it is much more efficient, nearly 10 times faster than HC and GA.

This type of empirical analysis can tell us the performance of each search algorithm, and whether there is any significant difference among them. However, what is the reason *why* we get those results? We try to shed light on this important research question in the next section.

5. INSIGHT KNOWLEDGE

Often, search algorithms are considered as "black boxes" in which only the output is considered. However, it is important to look *inside* those boxes to understand how search algorithms behave. Collecting this type of insight knowledge would help to obtain a better understanding of search base software testing, and it could lead to the discovery of new

Choose population K uniformly at random from S.
while global optimum not found
  Copy best n solutions from K to K'.
  while K' is not completely filled
    Select 2 parents from K according to selection criterion,
    Generate 2 offspring that are copies of their parents,
    Apply crossover on offspring with probability P_{xover},
    Mutate each offspring,
    Copy the 2 new offspring into K'.
  $K = K'$.

Figure 5: Pseudo-code of GA

Table 2: For each pair of search algorithms, p-values of Mann-Whitney U tests are shown.

	RS	HC	(1+1)EA	GA
RS	-	$8.31 \cdot 10^{-5}$	$2.20 \cdot 10^{-16}$	$2.17 \cdot 10^{-6}$
HC	$8.31 \cdot 10^{-5}$	-	$2.20 \cdot 10^{-16}$	0.28
(1+1)EA	$2.20 \cdot 10^{-16}$	$2.20 \cdot 10^{-16}$	-	$2.20 \cdot 10^{-16}$
GA	$2.17 \cdot 10^{-6}$	0.28	$2.20 \cdot 10^{-16}$	-

Table 3: For each fitness value, it is shown the number of solutions that have that value.

Fitness Value	Solutions
0	1008
1	832356
2	1514856
3	1503516
4	11669112
5	89892656

properties of the problem that could be exploited to design more efficient algorithms.

5.1 Fitness Values

The fitness function f can assume only 6 values, from 0 to 5. We evaluated all the over 105 million solutions, and for each fitness value we checked how many solutions have that particular fitness value. Results are summarised in Table 3.

Two interesting things can be noted. First, approximately only 1/6 of the solutions execute the method `fixAfterDeletion` (all solutions with fitness 4 or below execute it). Second, there is one global optimum only every 104,576.88 solutions. The global optima are only a small fraction of the search space S.

5.2 Global Optima

There are 1008 different global optima (i.e., test sequences that execute the target branch) in the search space S. By analysing them, we noted that they have a very precise structure. To simplify the analysis, we use the array M to represent the inputs in a test sequence. For example, $M[2] = 4$ means that the input of the third function call is a 4.

A global optimum follows all these rules:

- The test sequences are composed of 6 `put` followed by 1 final `remove`.
- The input values of the first 6 function calls are all different.
- The input value $M[6]$ for the last function call is a 0 if that value appears in the the first 6 function call inputs, otherwise it is a 1 (this latter happens only 144 times out of 1008).
- If $M[0] \in \{0,1\}$ and $M[1] \in \{0,1\}$, then $M[2]$,$M[3]$,$M[4]$, $M[5] \in \{2,3,4,5,6\}$. There are 240 global optima in which this condition appears.
- If $M[0] \in \{0,1\}$ and $M[1] = 2$, then there are 2 cases. If $M[2] \geq 3$, then $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\} \backslash M[2]$, otherwise $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\}$. The conditions of this rule appear in 96 global optima.
- If $M[0] \in \{0,1\}$ and $M[1] \geq 3$, then $M[2] \in \{0,1,2\}$. In particular, if is $M[2] = 2$, then $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\} \backslash M[1]$, otherwise $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\}$. There are 240 global optima in which this condition appears.
- If $M[0] = 2$ and $M[1] \in \{0,1\}$ there are 2 cases. If $M[2] \geq 3$, then $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\} \backslash M[2]$, otherwise $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\}$. The conditions of this rule appear in 96 global optima.
- If $M[0] = 2$ and $M[1] \geq 3$, then $M[2] \in \{0,1\}$ and $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\} \backslash M[1]$. There are 48 global optima in which this condition appears.
- If $M[0] \in \{3,4,5,6\}$, then $M[1]$,$M[2] \in \{0,1,2\}$. If in particular it is $M[1] = 2$, then $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\} \backslash M[0]$, otherwise $M[3]$,$M[4]$,$M[5] \in \{3,4,5,6\}$. The conditions of this rule appear in 288 global optima.

These rules have some interesting properties. On one hand, among the global optima the variables regarding the methods are independent. Each of them can in fact assume only one single value. On the other hand, the variables representing the inputs of the first 6 function calls are highly correlated to each other (i.e., their values depend on each other). The last input (i.e., $M[6]$) assumes the value 0 in 85.71% of the global optima, so it is less correlated than the others.

We can hence see a double structure in the global optima, in which half the variables are independent (regarding the global optima), whereas the other are highly correlated.

5.3 Expected Runtime of RS

Given 1008 global optima in a search space of 105,413,504 solutions, we can use the theorems in our previous work [2] to calculate the exact expected time for RS. Very simply, we can just divide the total number of solutions by the number of global optima. The calculation brings to the exact expected time of 104,576.88 fitness evaluations. The experiments shown in Table 1 give the result 82,570, that is roughly 20% smaller than the real expected time. More experiments (i.e., more than 100 runs) could bring to more precise results, but there would be no guarantee to obtain the exact value. We use theoretical analysis to obtain the real expected value, but to obtain the necessary knowledge of the number of global optima we used an empirical analysis, because it would have been too difficult to obtain it theoretically. That is a clear example in which theoretical and empirical analyses are combined together to obtain stronger results on a non-trivial problem.

5.4 Expected Runtime of HC

In the employed HC, the only randomised part are the starting points. Because the neighbourhoods have size 21 and because there are only 6 different values for the fitness function, we can infer that HC uses at most $1+((6-1)*21) = 106$ fitness evaluations before reaching either a local or a global optimum. This is a reasonable small number, hence it was possible to carry out an exhaustive empirical analysis of HC on every possible starting solution (but without considering the restarts, because there can be an infinite number of them).

We found that HC reaches a global optimum in 57,204 cases. On average, HC takes 27.12 steps when it reaches a local optimum, whereas 16.89 steps in the 57,204 cases in which a global optimum is found.

The restarts of HC can be theoretically modelled like a random search process [2]. Hence, we can combine this information with the average steps to reach an optimum that have been empirically calculated. That leads to the exact average time of 49,965.54 steps for HC to find a global optimum. The estimated mean in Table 1 is 46,770, and that is very close to the real average.

Like for RS, we combined empirical and theoretical analyses to get stronger results that would have been too difficult to obtain by using only one of them.

Considering that the neighbourhood size is 21, an average of 27.12 steps in case of local optima is quite informative. It in fact tells us that most of the search points are local optima. That is not a surprise if we note that there are only 6 possible values for the fitness function. Nevertheless, HC is on average twice as fast as RS, and that is not an hypothesis, it is a theoretically sound result.

5.5 Expected Runtime of (1+1)EA

To get more insight information on how (1+1)EA behaves in our search problem, we analysed what is the probability that a mutation on a solution can improve its fitness value. We divide the solutions based on their fitness values. For each group G_i with fitness value i, we calculate a probability for each group $G_{j<i}$ with lower fitness value j. Those represent the probability that a solution in group G_i can be mutated in a solution in group $G_{j<i}$. Unfortunately, this requires $|G_i| \cdot |G_{j<i}|$ evaluations, because we need to consider all the possible pairs among these two groups. For example, calculating the probability that a solution with fitness value 5 can be mutated to a solution with fitness value 4 would require $1.6 \cdot 10^{15}$ calculations.

We hence decided to estimate those probabilities by considering only 10^6 randomly chosen solution pairs for each pair of the groups (that are 15 in total). Table 4 summarises the results.

Why (1+1)EA is so much faster than HC? For HC most of the points are local optima. Therefore, HC spends most of its time in doing restarts even in the case of good quality solutions. For (1+1)EA it is quite difficult to mutate directly to a global optimum. However, Table 4 shows that there is a *gradient* toward the global optima if we go throughout all possible fitness values. For example, starting from G_5 jumping directly to a global optimum is done with probability of $2.44 \cdot 10^{-6}$. However, if we jump smoothly throughout all the possible fitness values (i.e., 5 transactions of group), the lowest probability we find is from G_1 to G_0, that is $3.58 \cdot 10^{-4}$, which is 146.72 times higher than jumping directly from G_5 to G_0.

If the probabilities in Table 4 were exact, we could theoretically calculate the exact expected time of (1+1)EA to find a global optimum [15, 16]. At any rate, we did that theoretical analysis. Comparing the result with Table 1 would in fact be useful to see if the two results are similar. Because the approach to obtain them is completely different, if they are similar that would give stronger support to their validity. The theoretical approach yields an expected time of 2,961 fitness evaluations, that is similar to the value 3,927 shown in Table 1.

Given $P(G_i,G_j)$ the probability that a solution in G_i can be mutated into one in G_j, and given $P(G_i)$ the probability that a random solution belongs to the group G_i, then the expected time E of (1+1)EA can be mathematically calculated with:

$$E = 1 + \sum_{i=0}^{5} (P(G_i) \cdot T(i)) \ ,$$

where $T(i)$ is calculated in this way:

$$T(i) = \begin{cases} 0 & \text{if } i = 0 \ , \\ \frac{1+\sum_{j=0}^{i-1}(P(G_i,G_j)\cdot T(j))}{1-P(G_i,G_i)} & \text{otherwise} \ . \end{cases}$$

Note that $P(G_i,G_j)$ are the probabilities shown in Table 4, whereas $P(G_i)$ can be easily calculated by dividing the cardinality of the input group (shown in Table 3) by the total number of possible solutions (that are more than 105 millions.) Also note that $P(G_i,G_j)$ is not a symmetric function (i.e., it is not necessarily true that $P(G_i,G_j) = P(G_j,G_i)$), that because (1+1)EA does not accept worse offspring.

5.6 Improved Fitness Function

Our analyses have shown that (1+1)EA has the best performance. But can we improve it even further? One way would be to improve the fitness function f (i.e., by making it smoother). Given the code in Figure 1, we see that the basic predicates on which f is based on are all boolean *flags*. They are:

- A <- `x != root`

Table 4: For each group G^y on axis y it is calculated the average probabilities that any of its solutions can be mutated to a solution belonging to a group G^x on axis x.

	G_0	G_1	G_2	G_3	G_4	G_5
G_0	1.0	0.0	0.0	0.0	0.0	0.0
G_1	$3.58 \cdot 10^{-4}$	0.99	0.0	0.0	0.0	0.0
G_2	$3.34 \cdot 10^{-6}$	$8.80 \cdot 10^{-3}$	0.99	0.0	0.0	0.0
G_3	$2.94 \cdot 10^{-5}$	$2.66 \cdot 10^{-2}$	$4.33 \cdot 10^{-2}$	0.93	0.0	0.0
G_4	$1.62 \cdot 10^{-7}$	$1.78 \cdot 10^{-3}$	$4.65 \cdot 10^{-3}$	$2.77 \cdot 10^{-3}$	0.99	0.0
G_5	$2.44 \cdot 10^{-6}$	$3.41 \cdot 10^{-3}$	$8.88 \cdot 10^{-3}$	$7.17 \cdot 10^{-3}$	$4.87 \cdot 10^{-2}$	0.93

- B <- `colorOf(x) == BLACK`
- C <- `x == leftOf(parentOf(x))`
- D <- `colorOf(rightOf(parentOf(x)) == RED`

These predicates directly depend on the internal state of the tree. The properties of the tree from which the predicates are calculated from are still flags. Therefore, flag removal techniques [6] would not help in this case.

Once the method `fixAfterDeletion` is executed, the first predicate to satisfy is $Z = A \wedge B$. The branch distance is based on that Z. Until the search for satisfying Z is not finished, any improvement that would make either C or D true is ignored. That because they are in nested branches that are not executed yet. One way to improve the fitness function is to consider these nested predicates even when they should not be executed yet [14]. Doing that is not always possible, that because side-effects and data dependencies could be present. In our case there is no problem in prematurely calculating C and D.

The branch distance of the `while` would be hence based on $A \wedge B \wedge C \wedge D$, whereas for the first `if` it would be $C \wedge D$ instead of just C. With this new "improved" fitness function, we carried out a set of experiments to see which amount of improvement can be achieved for (1+1)EA. We ran (1+1)EA with and without the improved fitness 1,000 times each. Results are shown in Table 5. The compared results are very similar, and a Mann-Whitney U test yields the p-value 0.56. So using this "improved" fitness function does not give any statistical difference.

Why we do not get any improvement? Having a look at the fitness value of each solution in the search space (as we did in Table 3) sheds lights on this problem. Results are shown in Table 6. If we compare Table 3 with Table 6, we can see that the only difference is that all the solutions that had a fitness value of 3 now they have a value of 2. Therefore, we should expect that this "improved" fitness function would be slightly *harmful*, because it makes the fitness function less smooth. In our problem, this happens for high values of the fitness, and those correspond to solutions from which it is relatively easy to escape from.

This is a clear example of an idea that sounds good, but that in some cases ends up to be harmful.

5.7 Tuning the GA

Most meta-heuristics have many parameters that need to be chosen. Often, no tuning is done, and the parameters are chosen from *common* values in literature. For example, in our experiments with GAs, we chose population size of 100, crossover rate of 0.75 and elitism rate of 1 individual per generation. Furthermore, several different selection mechanisms exist in literature.

Table 5: Comparison of (1+1)EA with and without the improved fitness function. Data were collected from 1,000 runs for each of the two configurations.

Improved f	Min	Median	Mean	Max	Variance
false	9	2522	3535	24190	$1.08 \cdot 10^7$
true	38	2604	3470	20220	$1.01 \cdot 10^7$

Table 6: For each fitness value of the "improved" fitness function, it is shown the number of solutions that have those values.

Fitness Value	Solutions
0	1008
1	832356
2	3018372
3	0
4	11669112
5	89892656

Unfortunately, an optimal choice of the parameters is problem dependent. We hence ran a new set of experiments in which we tested 10 population sizes from 2^1 to 2^{10}, then 10 crossover rates from 0.0 to 0.9 with 0.1 of increment. Finally, we considered 10 elitism rates from 0% to 90% (with increments of 10%) of the current population. We tested all their combinations, i.e. 1,000 configurations. For each configuration we ran 100 independent experiments, and we calculated the average number of fitness evaluations.

Because for some configurations the GA becomes very inefficient, we had to put the stopping criterion of limiting the fitness evaluations by an upper bound of 1 million evaluations.

The best configuration we found does have a population size of 256, crossover rate of 0.9 and elitism rate of 0.6. Because there is an infinite number of possible configurations of the parameters, we cannot guarantee that these values are optimal.

Because these values are different from the ones we originally used, we hence decided to compare again GA versus (1+1)EA with new experiments. We do that in order to assess whether (1+1)EA is still better. We ran each algorithm 1,000 times. Results are shown in Table 7. Note that we

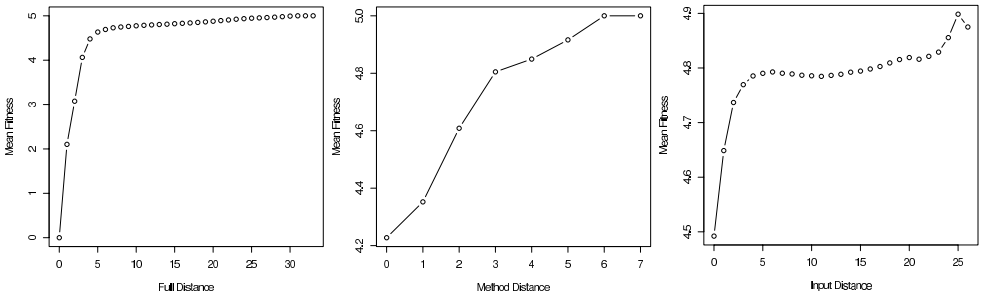

Figure 6: Given three different types of distance, each plot shows the correlation between the distance from the closest global optimum and the average fitness value.

are not re-using the values in Table 5. Mann-Whitney U test yields the p-value $2.2 \cdot 10^{-16}$. This means that there is strong evidence that their median values are different.

Why (1+1)EA is better than GA? This could happen if the search landscape is like a "stair", whose bottom represents the global optimum. Solutions on the same step have same fitness, i.e. the steps are plateaus. It would be easier to climb down that stair with a single individual instead of an entire population (under the assumption that crossover is not very useful). *OneMax* (one of the simplest and most used toy problem in evolutionary computation) does have that landscape, and indeed there are theoretical results that show that the population-based evolutionary algorithms analysed so far are not better than (1+1)EA [10, 19].

To give feasible explanation for the behaviour of GA in our problem, we analyse whether our problem has a similar type of stair structure. For each possible solution we calculated its fitness value and its closest distance from a global optimum. We used the Hamming distance in three different ways: on the full sequence (14 variables), on only the method types (7 variables) and finally on only the inputs of the methods (7 variables). For each possible distance value, we calculated the average fitness value that the solutions with that distance have. Results are shown in Figure 6. Indeed, a stair structure is clearly evident if we consider the average fitness. That is particularly true for the method types, but not if we only consider the inputs of the methods. This could be related to the correlation of the variables we discussed in Section 5.2.

The presence of this stair structure is only a hypothesis (although well supported) because we are considering the average fitness. The presence of more complex dynamics cannot be excluded. A first step to shed light on this matter is to show how the performance changes with the modification of the three tested parameters. Figure 7 plots the number of average fitness evaluations when one of the three parameters is fixed to its best found value and the other two vary.

First, from Figure 7 we can see that the crossover operator has a positive influence on the performance. However a more detailed analysis of this operator will be done in future work. Second, we can note that when no elitism is used, the performance is poorer. Finally, the role of the population size is different from what we did expect. Considering the very good performance of (1+1)EA, we were expecting that smaller populations would perform better than larger ones.

Table 7: Comparison of (1+1)EA against tuned GA. Data were collected from 1,000 runs for each algorithm.

Algorithm	Min	Median	Mean	Max	Variance
(1+1)EA	27	2439	3513	26520	$1.14 \cdot 10^7$
GA	294	3730	7883	466100	$4.39 \cdot 10^8$

In fact, for small populations we were expecting that GA would be very similar to (1+1)EA. What we see in Figure 7 is exactly the opposite.

A more detailed analysis of our GA implementation showed us that our elitism operator tends to prefer the same best individuals among the solutions with same fitness. That because in the implementation we just ascendingly sort the population and we take the first n best starting from left. The new offspring are added after these n best individuals. The sort algorithm we used is in the Java API 6, and it preserves the order of equivalent elements. We thought that was a reasonable implementation, but it turned out that for small populations it prevents *random walks* on the plateaus, that it is a well known problem [11]. This is a clear example of a low level detail of a search algorithm that initially does not seem to be very important, but that then it results to have drastic impact on the performance. Future work will investigate this matter in more details.

6. CONCLUSIONS

In this paper we deeply analysed a difficult software testing problem. We carried out both theoretical and empirical analyses. Our goal was to give insight knowledge of the problem to be able to answer *why* we obtain those particular behaviours of the analysed search algorithms. If we know how search algorithms actually work inside their black boxes, then we can exploit this information to understand which are their limitations and therefore how we should proceed to improve them.

We found out that a simple (1+1)EA performs better than a fairly tuned GA. This is in contrast with the current literature, in which population algorithms like GAs are very common, and single individual algorithms are considered not able to cope with the difficulty of software testing. But in contrast to many empirical analyses in literature, we gave sound reasons supported by compelling evidence to *explain* such a behaviour.

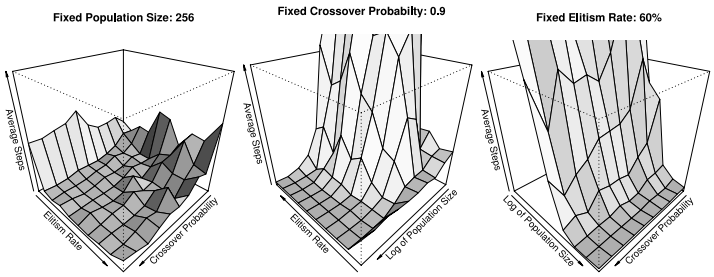

Figure 7: Average fitness evaluations when one of the three parameters (i.e., population size, crossover probability and elitism rate) is fixed to its best found value and the other two vary.

We used constraints on the input ranges and on the number of method calls in the test sequences. On an unknown testing problem, these constraints could preclude the finding of any global optimum. However, in our case even a RS search can give optimal solutions in a reasonable time when constraints are used. Therefore, the problem is not so difficult. For future work, it will be important to extend our analyses to scenarios in which no constraint is used. That would help to understand the impact of the choice of these constraints.

7. ACKNOWLEDGMENTS

The author is grateful to Pietro Oliveto for insightful discussions. This work is supported by an EPSRC grant (EP/-D052785/1).

8. REFERENCES

[1] A. Arcuri. Full theoretical runtime analysis of alternating variable method on the triangle classification problem. In *International Symposium on Search Based Software Engineering (SSBSE)*, 2009. (to appear).

[2] A. Arcuri, P. K. Lehre, and X. Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *International Workshop on Search-Based Software Testing (SBST)*, pages 161–169, 2008.

[3] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, 1990.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[6] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[7] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 73–83, 2007.

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.

[9] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[10] T. Jansen, K. A. D. Jong, and I. A. Wegener. On the choice of the offspring population size in evolutionary algorithms. *Evolutionary Computation*, 13(4):413–440, 2005.

[11] P. Lehre and X. Yao. Crossover can be constructive when computing unique input output sequences. In *Proceedings of the International Conference on Simulated Evolution and Learning (SEAL)*, pages 595–604, 2008.

[12] P. K. Lehre and X. Yao. Runtime analysis of (1+1) ea on computing unique input output sequences. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1882–1889, 2007.

[13] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[14] P. McMinn and D. Binkley. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop*, pages 165–182, 2005.

[15] M. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[16] P. S. Oliveto, J. He, and X. Yao. Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 4(3):281–293, 2007.

[17] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[18] D. Whitley. The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, pages 116–121, 1989.

[19] C. Witt. Runtime analysis of the (μ+1) ea on simple pseudo-boolean functions. *Evolutionary Computation*, 14(1):65–86, 2006.